Solution Adapted Mesh Refinement and Sensitivity Analysis for Parabolic Partial Differential Equation Systems*

Shengtai Li[1,2], Linda R. Petzold[1], and James M. Hyman[2]

[1] University of California, Santa Barbara, CA 93106, USA
[2] T-7, Los Alamos National Laboratory, Los Alamos, NM 87544

Abstract. We have developed a structured adaptive mesh refinement (SAMR) method for parabolic partial differential equation (PDE) systems. Solutions are calculated using the finite-difference or finite-volume method in space and backward differentiation formula (BDF) integration in time. The combination of SAMR in space and BDF in time is designed for problems where the fine-scale profile of sharp fronts in space should be resolved and implicit integration in time is necessary to improve the efficiency of the computation. Methods for forward sensitivity analysis on the adaptive mesh are presented.

1 Introduction

Structured adaptive mesh refinement (SAMR) has been used extensively to solve partial differential equations (PDEs) [3,9,13]. SAMR uses a hierarchical block data structure where each block (called *patch*) can be solved as a single grid. Most implementations of SAMR have used an explicit time integration, and refined time as well as space by taking local smaller time steps for finer grids. The time stepsize for explicit integration is limited by the CFL condition. Explicit methods are appropriate for hyperbolic systems, where the CFL number is proportional to $\Delta t/\Delta x$. However, for a parabolic system the CFL number is proportional to $\Delta t/(\Delta x)^2$. Hence the time step for an explicit integration needs to be very small to ensure stability. It is desirable to solve this type of problem with implicit time integration. Implicit time integration is also preferred for solving steady-state and slow-transient problems, because the stepsize restrictions are less stringent than for explicit schemes (there may not be any). Although SAMR has been available for more than a decade, its combination with implicit time integration is still in its infancy. Before describing our algorithm and implementation, we first discuss two available adaptive grid implementations for parabolic PDE systems.

Verwer et al. [15] designed a local uniform grid refinement (LUGR) with second order backward differentiation formula (BDF) method in time. Instead

* This work was partially supported by DOE contract number DE-FG03-00ER25430, NSF grant CCR-9896198, NSF/ARPA PC-239415, and NSF ACI-0086061.

of using a hierarchical block data structure, LUGR uses a data structure specially designed for its algorithm. A standard second-order finite difference is used in the spatial discretization, central on the internal domain and one-sided at the boundaries. A fixed second-order two-step implicit BDF method with variable stepsizes is used for the time integration. The resulting system of nonlinear equations in each time step is solved by a modified Newton method and (preconditioned) iterative linear solver. LUGR is not flexible with respect to changes in the spatial discretization or time integration method.

Flaherty et al. [8] designed an adaptive overlapping grid (AOG) method using Galerkin's method with a piecewise polynomial basis in space and a singly implicit Runge-Kutta (SIRK) integration method in time. A tree-based hierarchical data structure is used. The grid refinement strategy is based on error estimates for Galerkin's method. Due to the frequent stops and starts that are needed in conjunction with the refinement/coarsening process, AOG opted for single-step SIRK methods for the time integration. Unfortunately, these methods often proved to be more costly than a multistep method. The integration of each local patch is done separately, and a Schwarz alternation iteration is used to obtain satisfactory accuracy in overlapping regions. AOG also uses overlapping and rotated grids in the refinement processing, which proved to be disadvantageous for general problems.

In this paper, we study how to combine the SAMR method with variable order and variable stepsize BDF time integration. The BDF methods are implicit multistep methods for solving systems of ordinary differential equations (ODEs) or differential-algebraic equations (DAEs). An efficient BDF code DASSL, developed by Petzold [4], has been widely used. In DASSL, the implicit system is solved by a modified Newton iteration. The linear system at each Newton iteration is solved by a dense or banded direct solver. An extension of DASSL, DASPK, was developed by Brown, Hindmarsh and Petzold [5]. DASPK has an additional option of using a preconditioned incomplete GMRES method to solve the linear system at each Newton iteration, which is particularly effective in the method of lines (MOL) solution of time-dependent PDEs in two or three dimensions.

Sensitivity analysis is important in many engineering and scientific applications. The information contained in the sensitivity trajectories is useful for parameter estimation, optimization, model reduction and experimental design. A DASPK package (DASPK3.0) with forward sensitivity analysis capability was developed by the authors [11]. DASPK3.0 also incorporated many other new features for the efficient integration of the ODE/DAE.

Compared with explicit integration, applying implicit time integration with an AMR system has a lot of difficulties. First, each patch cannot be integrated independently in one time step, because it may share boundary points with other patches. Second, the implicit system needs to be solved in each time step. Therefore at least one linear system must be solved in each time step. How to solve the linear system efficiently is essential to the success of the PDE solver. Related issues of how to reuse the previously-evaluated

Jacobian or preconditioner for the current time step, and how to get the initial guess for the Newton iteration are also important. Third, it is more difficult to take a local time step for the finer grid for an implicit method than for an explicit method. The time stepsize at each level is usually determined by accuracy instead of stability for the implicit method. Thus, the ratio of the stepsizes for different levels usually is not an integer(which is required for the hierarchical AMR data structure). There are some other difficulties, such as storing and updating the Jacobian or other linear system information for each level/patch if it is solved separately, restarting the integration after each refinement, etc.

In the following sections, we propose some strategies to overcome or circumvent these difficulties. We also study how to efficiently compute the sensitivities of adaptive solutions for the PDEs. Difficulties related to adaptive data structure and discretization are addressed.

2 AMR with Hierarchical Block Structure

Our SAMR method makes use of the AMR data structure and refinement strategy in [9]. In order to be read independently, we outline the method here.

2.1 Hierarchical block structure

To efficiently manage the data on each level in the AMR algorithm, the points are grouped (clustered) into logically-rectangular blocks called patches. These patches are the building blocks for the hierarchical grid structure and are the basic data unit for refining the grid in space. A patch is treated as a single grid with all the attributes of a single grid.

We use an indexed linear array hierarchical data structure in [9]. The hierarchical grid data structure $G=|n|G_1|G_2|...|G_n|$ contains the number of levels of the grid and pointers to the grid on each of the lower levels. The data structure on the i-th level $G_i=|m_i|p_1|G_{i,1}|p_2|G_{i,2}|...|p_{m_i}|G_{i,m_i}|$ contains information on the patches, where m_i denotes the number of patches. The data structure $G_{i,j}$ contains information for the j-th patch on the i-th level. For a 1-D grid, the pointer p_j is the index of the patch in the coarse grid that is the parent of the patch $G_{i,j}$, For a 2-D and 3-D grid, the variable p_j contains the number of parent coarse grids for the patch $G_{i,j}$. An auxiliary array is used to store the indices of the parent grids.

2.2 Refinement strategy

The core of the AMR algorithm is in choosing how to cover the regions that need refinement with a finer grid. The *Remesh* stage to cover subdomains with higher resolution patches is the most algorithmically complex AMR operation in the refinement process. The remesh stage is divided into two

```
Remesh(level)
begin
    maxlevel = the maximum level allowable, flevel = the finest level existing;
    flevel = max(flevel + 1, maxlevel);
    while (flevel-1 needs no refining) decrease flevel by 1;
    // Readapt the current grid
    for slevel = flevel-1 downto level do
        Refine(slevel)
            Select (slevel): flag the inaccurate points which need refining;
            Expand (slevel): add buffer zones around the flagged region;
            Cluster (slevel): group the flagged points into clusters;
    for slevel = level upto flevel-1 do
        Regrid (slevel +1): define the solution values for the readapted grid;
    // Refine to generate new finer grid
    while (flevel < maxlevel and flevel needs refining) do
        Refine (flevel);
        Regrid (flevel + 1);
        increase flevel by 1;
end
```

Fig. 2.1. Pseudo-code for AMR remeshing algorithm.

processes (see Fig. 2.1): *readapt* (including refine and coarsen) the current grid and *refine* to generate a new finer grid. Both processes have two small steps (see Fig. 2.1): *refine* and *regrid*.

The readaptation must be designed to capture the features that appear in the finer levels but would not be identified if the process started with the solution on the coarsest grid and then adapted the grid to the finer levels. Therefore, we initiate the mesh readaptation on the finest level possible. Note that this is different from the local uniform grid refinement (LUGR) method [15]. This grid is then coarsened or refined based on the selection algorithm. This process continues until all of the indicated levels have been readapted. The regridding step (see Fig. 2.1) (defining the solution values for the readapted grid) is done in reverse. It starts from the coarsest level possible. After the first process, if the finest level available does not reach the maximum level allowable and needs further refinement, we start the second process to refine and generate finer level patches.

We adopt the monitor function proposed by Verwer et al. [15] to identify the regions to coarsen or refine. The monitor function is defined for each grid point (i, j). We initiate a level refinement if there is a point where the monitor function exceeds the tolerance. In order to ensure proper nesting, if the current level grid has a grandparent, those points are also flagged. To be flexible, our software has an option to allow users to provide monitor functions.

For some applications, the monitor function may fail to identify all the regions that need to be refined. Also, there are situations where we may only be interested in the final steady state solution. In these cases, the efficiency can be improved if the user has control over the AMR process. In an extreme situation, a user may want complete control to guide the refinement process at any time and any place.

We incorporate several options in our software for user control of the refinement. The user can force a refinement through a grid file and modify the refinement parameters at any time [9].

3 Time Integration

It would be appealing to take a local time step for a local finer grid, as is done in conjunction with explicit time integration [9]. However, due to the difficulties mentioned in Section 1, we decided to synchronize the time step for all the grids. In fact, for a parabolic problem solved by an implicit method, the time step is determined by accuracy rather than stability considerations, and the difference in stepsize between grids in different refinement levels is generally small.

As pointed out in [15], the solution injected from a finer grid is in general not a solution of the PDE system discretized on a coarser grid and hence can cause convergence problems in the Newton iteration if it is used as the initial guess. Therefore, we solve the whole AMR system simultaneously. That is, the entire AMR hierarchical structure is transformed into one linear structure used by the implicit time integration solver.

To design software that is easy to use by an application scientist, we use the method of lines (MOL) approach. That is, the PDEs are first discretized into ODEs/DAEs, and then existing ODE/DAE software is used in the time integration. Our implementation has no restriction on the spatial discretization and time integration solver. We use DASPK3.0 for the time integration because of its capabilities for DAEs, implicit time integration and sensitivity analysis. In the next subsection, we describe the transformation between the DASPK3.0 and AMR data structures.

3.1 Transformation between DASPK and AMR data structures

The hierarchical data structure of AMR provides us a possibility to integrate each level or patch separately. However, the difficulty of such a separate computation is synchronization of the time step. Because the time stepsize is computed inside the time integration solver (DASPK in our case), we cannot expect that different levels or patches would use the same time stepsize. Another problem with separate computation is that the Schwarz alternation iteration must be used.

To avoid these complications and difficulties, we integrate the whole system as one big ODE/DAE system. Thus, we must transform the hierarchical

data structure into a flat structure that can be used by DASPK. In order that the equations or residuals can be evaluated patch by patch in the AMR hierarchical system and the solutions visualized easily, the transformation must also be done in the reverse.

To eliminate redundancy and inconsistency, we require that any point or cell in the AMR system be evaluated only once. The transformation is illustrated in Fig. 3.1 and is designed as follows. Beginning with the finest level, each point in each patch in a level is copied to a linear array and marked after it is copied; if a point is marked by a previous patch or level, it is skipped. This process is done level by level until the base grid is finished. The inverse transformation is a little more complex. After the inverse copying from the linear array to the AMR hierarchical structure is done, the uninitialized points in the AMR hierarchical structure are collected by copying from the sibling grid and finer children grids.

For the inverse transformation, the ghost boundaries are also required in evaluation of the equations or during refinement. The ghost boundaries for each patch must be collected from three sources. First they are calculated from external boundary conditions if any of its boundaries reaches the external boundary. Then they are copied from the sibling internal grid points. Finally, if there are still uninitialized ghost boundary points, interpolation from the parent coarse grid is used.

3.2 Warm restart after refinement

After a new grid is generated and the solution has been interpolated from the old mesh to the new one, the simplest approach would be to restart the time integrator as though solving a new problem. This is called a *full restart* by Berzins et al.[2] and is appropriate for single-step time integration methods such as SIRK. For multistep methods, a full restart would cause the ODE/DAE solver to choose the lowest order single-step method and to reduce the time step size to satisfy the error tolerance of the lowest order method.

In a warm restart (or *flying restart* [2]) the history array used by the ODE/DAE solver is also interpolated to the new mesh, and the integration is continued with almost the same step size and order as would have been used had the remeshing not taken place. Because the number of equations may have changed during the remeshing and the Jacobian matrix (preconditioner in our case) is difficult to interpolate accurately, we always reevaluate the Jacobian matrix in a warm restart.

Compared with a global rezone method [10], the interpolation from an old grid to a new one in AMR involves less error, because the most interesting portion of the fine grid overlaps with that of the old grid, and the overlapping part can just be copied from the old mesh. If the refinement in the AMR system is timely, the interpolation occurs only near the coarse-fine interface, where the new grid points are generated by the refinement process. Since the

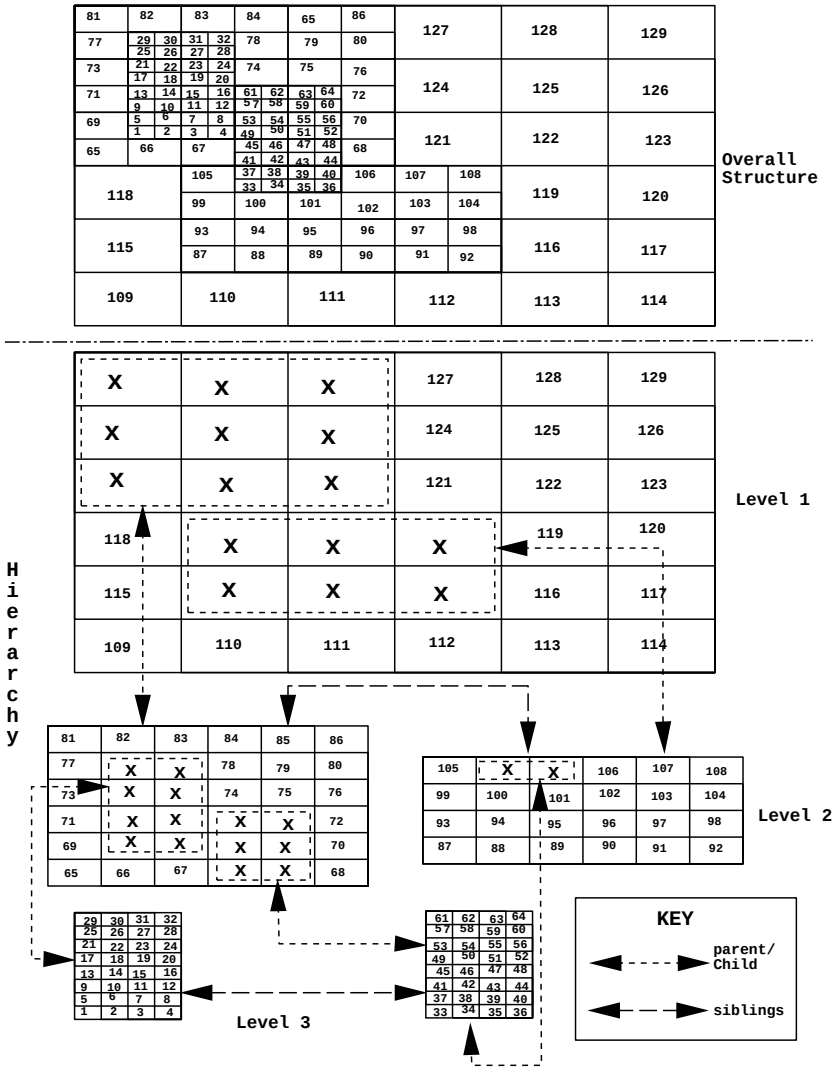

Fig. 3.1. The hierarchical data structure for AMR and its transformation to the DASPK flat structure. The number inside the mesh is the order number in the DASPK flat structure. The cells with "**x**" inside have been defined by other patches. A total of 129 cells is copied from the AMR hierarchical data structure to the DASPK flat structure.

discretization error near the coarse-fine interface is generally much smaller than at the internal points, linear interpolation is sufficient in most cases. When a new grid has more refinement levels than the old one, bivariate cubic interpolation can be applied.

The interpolation errors in a warm restart may not be sufficiently small and may cause the ODE/DAE solver to reduce the stepsize and/or order. In our experience, even if the ODE/DAE solver eventually restarts with the first-order method, the time-step size is much larger than that for a full restart.

3.3 Reducing the overhead related to mesh adaptation

Even with the warm restart, the overhead of the mesh adaptation is relatively high. The most significant cost is evaluation of the Jacobian. The adaptation process (including refining and regridding steps) and interpolations for the history array actually take less time than the evaluation of the Jacobian. Since the Jacobian must be evaluated after each adaptation, it is important to reduce the number of adaptations and/or reduce the computational work of evaluation of the Jacobian.

The number of adaptations is determined by the number of time steps k_{amr} between two adjacent refinements. k_{amr} is affected by many factors. One of them is the number of buffer zones (k_{buf}) added during the refinement. We note in our experiments that if k_{buf} increases, k_{amr} can be larger. However, when k_{buf} increases, the finer grid becomes larger and requires more time to solve. In our numerical tests, we found that $k_{buf} = 2$ and $k_{amr} \approx 10$ have good performance for most problems. How to choose k_{amr} dynamically, as we did in [10] in the case of explicit integration, is under investigation.

Replacing with the new grid adaptively An alternative approach to reduce the overhead of the adaptations is to replace the old mesh with the new adaptive one only when the variance is big enough. A fixed k_{amr} can be taken for this approach.

As we have mentioned, the new adaptive mesh shares the most interesting part with the old one. In our implementation, we calculate the number of grid cells shared by both the new grid and the old one after each adaptation. A *ratio* that measures the shared percentage is calculated as follows,

$$ratio = \frac{2 * N_{share}}{N_{old} + N_{new}}, \tag{1}$$

where N_{share} is the number of shared grid cells, N_{old} is the number of cells in the old grid and N_{new} is the number of cells in the new adaptive grid. The old mesh is replaced by the new one only if $ratio < 0.92$. This number was determined experimentally to give the best performance over a wide range of problems. Otherwise, the old mesh is used as if no adaptation has

occurred. This strategy can sharply reduce the number of warm restarts and the number of Jacobian evaluations.

ILU preconditioner and ADIFOR To evaluate the Jacobian efficiently, we can choose a simple preconditioner that requires less computational work, such as a block-diagonal or block-Jacobi preconditioner. However, the performance for these simple preconditioners was not very promising during our numerical experiments. We opted for an incomplete LU (ILU) factorization preconditioner [14] in our software.

For an ILU preconditioner, the Jacobian should be evaluated and stored in each evaluation of the preconditioner. Unlike a single nonadaptive grid, the bandwidth of the Jacobian in our system can be very large because the solution in one patch/level can be related to the solution in another patch/level and they can be far away in different locations in the flat DASPK structure after transformation. Note that in Fig. 3.1, the cell at position 33 relates to the cells at positions 100 and 105.

The cost of evaluating the Jacobian J (C_{eval}) via ADIFOR, if the sparse forward mode is used, is related to the cost of residual evaluation (C_{fun}) by $C_{eval} \simeq a \cdot m \cdot C_{fun}$, where $a = 3$ for the basic forward mode of automatic differentiation, and m is the maximum number of nonzero entries in any row of the Jacobian. However, when the finite-difference method is used, the cost is $C_{eval} \simeq b \cdot C_{fun}$, where b is the bandwidth of the Jacobian matrix. For a PDE in a 2-D domain, m is usually small ($m = 5$ if central difference in space is used for a scalar PDE) but b is large. Therefore, we recommend using ADIFOR to evaluate the Jacobian whenever possible.

Reducing the computation of data structure transformation The transformation between the AMR hierarchical data structure and the DASPK linear structure is frequently used during function evaluations, mesh refinements and warm restarts. To reduce the overhead of the adaptations, we must reduce the computation time of each interface transformation.

We can go through the matching process proposed in Section 3.1 each time to do the transformation or inverse transformation. Noting that the matching process is the same if the mesh does not change, we propose to do the matching process only once for a new mesh.

The algorithm is as follows. When a new mesh is generated, we go through the AMR hierarchical data structure and the DASPK linear structure and match them one by one using the algorithm of Section 3.1. During the processing, we can take advantage of the indexed linear array implementation for the hierarchical data structure. We actually do the matching between two linear structures instead of one hierarchical (tree) structure and one linear structure. The solutions in the AMR hierarchical data structure contain data at the ghost boundaries and shared internal points among different patches. Hence there are more of them than those in the DASPK linear structure.

Two index arrays are used to store the locations of the solutions in each linear structure during the matching process. One is used to store the indices of the elements in the AMR structure for the elements in the DASPK structure. The other is used to store the indices of the already marked elements in the AMR structure and their corresponding elements in the DASPK structure.

For the transformation from the AMR structure to the DASPK structure, we use only the first pointer array. For the inverse transformation, both pointer arrays are used. The ghost boundary data for the AMR structure, if needed, are collected separately. The transformations can be done much faster with the help of the two arrays. Since these two arrays are computed only once for a new mesh, the total computational efficiency is improved.

4 Sensitivity Analysis for PDEs

DASPK3.0 has a capability for forward sensitivity analysis[11]. The sensitivity equations for the DAEs have many good properties which can be taken advantage of. First they are linear with respect to the sensitivity variables. Second, the Jacobian matrix for the sensitivities is the same as for the original DAEs. We would like to make use of this sensitivity analysis in the SAMR solution of PDEs.

4.1 Sensitivity ODEs vs. sensitivity PDEs

There are two possibilities for evaluating the sensitivity residuals of a PDE system. First, we can use the MOL approach and transform the PDE system into an ODE/DAE system. Then the sensitivity methods in DASPK3.0 can be used. The sensitivity equations can be evaluated by several options in DASPK3.0, such as the finite-difference or ADIFOR options. This approach does not require any modification of the PDE discretization codes.

The other approach is to solve the sensitivity PDEs coupled with the original physical PDEs directly. They are simultaneously discretized in space and then the coupled ODE/DAE system is solved by DASPK3.0. Similar to the sensitivity DAEs, the sensitivity PDEs are linear with respect to the sensitivity variables. Since the PDE system is usually much simpler before discretization, the sensitivity PDEs can be easily obtained. Some special discretizations or transformations used for the state PDEs can be reused in the sensitivity PDEs. Therefore, it is usually more efficient and accurate to evaluate the sensitivity equations by this approach than by the first approach. In fact, we have found that if a nonlinear spatial discretization scheme (e.g. upwinding scheme) was used, the first approach might produce incorrect sensitivities.

For an implicit solver like DASPK, the cost of the Newton iteration for solving the nonlinear system of equations often dominates the computation. It is easy to solve the coupled system without distinguishing the state and

sensitivity variables in the second approach. However, it is much more efficient if we evaluate the Jacobian/preconditioner only for the state variables, and reuse them in solving for the sensitivities. DASPK3.0 has an option for the user to input the residual for the state and sensitivity equations respectively. Distinguishing the state and sensitivity variables in DASPK also allows the user to exclude the sensitivity variables from the stepsize control, which in our experience has led to better performance as well as accurate sensitivities.

For an adaptive grid solver, we must decide whether the selection of mesh refinement should be based only on the state PDEs or on both the state and sensitivity PDEs. We observed in our applications that the sensitivity PDEs shared the same refinement regions as the state PDEs. Therefore, we prefer excluding the sensitivity equations from the monitor function evaluations for efficiency considerations.

4.2 Sensitivity analysis with AMR hierarchical structure

In DASPK3.0, the sensitivity variables are stored separately right after the state variables, whereas a sensitivity variable is taken as a PDE variable and all of the variables in a patch are stored together in the AMR data structure. This causes some difficulty in transformation from the AMR hierarchical data structure to the DASPK flat structure and in the residual evaluations of DASPK.

The transformation between the AMR and DASPK data structures proceeds in two steps. In the first step, we do the transformation only for the state variables. In the second step, we transform one by one for the sensitivity variables. If the sensitivity variables are not needed, the second step is skipped. In sensitivity analysis using DASPK3.0, the Krylov iteration uses only the residual evaluations of the state variables. The number of residual evaluations for the state variables is much larger than that for the sensitivity variables. Therefore, the overhead in the transformation can be much reduced by the two-step technique.

5 Numerical Experiments

In this section, we give an example to illustrate the effectiveness of our algorithm and software. The number of steps k_{amr} between two adaptations is chosen to be 6. The refinement ratio is chosen to be 2, and the number of buffer zones (k_{buf}) is 2 unless it is specified otherwise. Central-differencing discretization in space is used. The error tolerance in DASPK is chosen to be RTOL=ATOL=10^{-5}. All of our computations are done in double precision on a 450HZ PC with the Linux operating system. For comparison, we give some key statistics of our computation.

NWR	Number of warm restarts
NTS	Number of time steps
NRE	Number of residual evaluations
NJE	Number of Jacobian evaluations
NETF	Number of error test failures
MXEQ	Maximum number of equations in DASPK format
CPU	Total CPU time taken to solve the problem

5.1 SAMR solution

This example of reaction-diffusion type is described in Zegeling [16]. The PDE is given by

$$u_t = \Delta u + D(2-u)\exp(-d/u), \quad \text{on the domain } \Omega = (0,1)\times(0,1) \quad (2)$$
$$u|_{t=0} = 1, \qquad \frac{\partial u}{\partial n} = 0, \text{ at } x = 0, y = 0, \quad \text{and } u = 1, \text{ at } x = 1, y = 1,$$

where Δ is the Laplacian operator and $D = Re^d/d, R = 5, d = 20$. We output the solution at $t = 0.30$. Since the solution before $t = 0.25$ is very smooth, we turned off the refinement and used only the base grid before $t = 0.25$. The tolerance for grid refinement was chosen to be TOLS=0.001.

The full restart is extremely slow. The warm restart, however, is much faster. We note that after refinement the solver DASPK3.0 uses almost the same order as before the refinement. The first refinement takes place at about $t = 0.25$. We chose $k_{amr} = 8$ for the later refinements. The comparison for methods with different refinement levels and different buffer zones are shown in Table 1. The contour plots and refinement patches are displayed in Fig. 5.1.

Table 1. Comparison of different methods for flame propagation problem.

Base grid level		k_{buf}	NWR	NTS	NRE	NJE	NETF	MXEQ	CPU
201×201	1	N/A	0	147	687	15	18	40401	155
101×101	2	2	17	188	742	37	22	17450	56
51×51	3	2	18	209	802	38	29	12637	32
51×51	3	1	19	220	854	41	32	11557	31

The warm restart does have some adverse effect on the time step selection, which is shown in Fig. 5.1-c. We suspect that this is due to the interpolation errors from the old grid to the new grid.

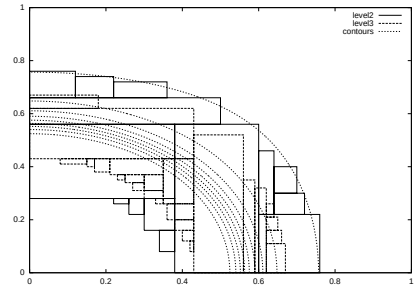

Fig. 5.1-a. AMR with 50×50, 3 refinement levels and 2 buffer zones. 9 contours between 1.1 and 1.9 are plotted.

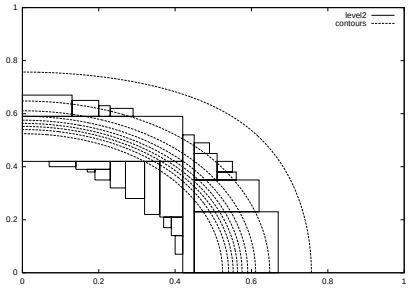

Fig. 5.1-b. AMR with 100×100, 2 refinement levels and 2 buffer zones. 9 contours between 1.1 and 1.9 are plotted.

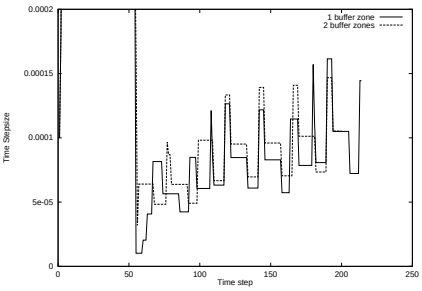

Fig. 5.1-c. The warm restart can cause the DAE solver to reduce the time step after each refinement. Three-level refinement and a 50×50 base grid is used.

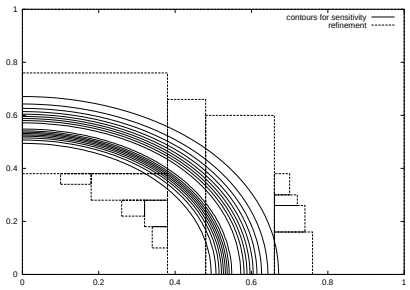

Fig. 5.1-d. Contour plots for the sensitivity with respect to R. 9 contours between 2 and 18 are used. The contours increase from both sides to the middle of the refinement.

Figure 5.1: Results for flame propagation problem.

5.2 Sensitivity analysis

We also computed the sensitivity with respect to the parameter R in Eq. (2). The sensitivity PDE is given by

$$s_t = \Delta s + D/R(2-u)\exp\left(-\frac{d}{u}\right) - sD\exp\left(-\frac{d}{u}\right)\left(1-\frac{d(2-u)}{u^2}\right). \quad (3)$$

We used two refinement levels and a 51×51 base grid. The error tolerance for the sensitivity variables was the same as that for the state variables. The partial error test (excluding the sensitivity variables from the error test) and staggered corrector method option was used in DASPK3.0. The other parameters were the same as those without considering sensitivity.

Two options for sensitivity evaluation in DASPK3.0 were tested: input analytically and by ADIFOR with seed matrix. They produced the same

results, which is not surprising since no special technique is used during the spatial discretization. The efficiency for the two options was almost the same. We also solved Eqs. (2) and (3) without using the sensitivity techniques of DASPK3.0 (see "taken as PDEs" method in Table 2). In this method, we cannot exclude the sensitivity variables from the error test in DASPK3.0, and the warm-restart after each refinement does not work well after $t = 0.29$. We suspect the reason is that the sensitivity changes too rapidly. We should mention that the accuracy of the sensitivity did not improve much by including the sensitivity variables in the error test. The contour plots are almost the same.

Table 2. Comparison of different methods for sensitivity analysis of flame propagation problem. *This number includes evaluations of the sensitivity equations.

Sensitivity Evaluation Method	NST	NRE	NJE	NETF	CPU
Input analytically	191	3817	39	24	79
ADIFOR with seed matrix	191	3817	39	24	82
Taken as PDEs	1315	4336*	491	211	329

6 Conclusion

We have presented our implementation of AMR with the implicit DAE solver DASPK, for parabolic problems where implicit time integration is best suited. Several difficulties have been described when AMR is combined with implicit integration. We have provided some strategies to overcome and/or circumvent the difficulties. Numerical results demonstrate that these strategies are effective.

We have also discussed how to combine the sensitivity analysis with the AMR hierarchical data structure. An interface was designed between the AMR hierarchical structure and the DAE solver flat structure to facilitate the use of the DAE solver and data visualizations.

A large-scale sparse linear system must be solved in the implicit time integration. How to improve the efficiency of the linear solver is key to the success of our combination of AMR and DASPK. We provide an ILU preconditioner which is evaluated by ADIFOR. Other kinds of preconditioners, such as additive Schwarz alternation preconditioners, are under investigation.

Although the warm restart technique can greatly improve the efficiency, the restart/interpolation process still has an adverse effect on the time step selection. We think this is due to the interpolation error after the refinement. How to improve the time step selection after each restart is an open problem.

The forward sensitivity method described in this paper is attractive when there are relatively few sensitivity parameters. When a large number of sen-

sitivity parameters and only a few derived functions are involved, the adjoint sensitivity method may be more advantageous. With the help of the recent results on the adjoint method for DAEs [7,6], we have studied several theory and implementation issues for the adjoint sensitivity method on an adaptive grid for partial differential-algebraic equations (PDAE) [12].

References

1. C. Bischof, A. Carle, G. Corliss, A. Griewank and P. Hovland, *ADIFOR–Generating derivative codes from Fortran programs,* Scientific Programming (1992).
2. M. Berzins, P. J. Capon and P. K. Jimack, On spatial adaptivity and interpolation when using the method of lines, *Appl. Numer. Math.*, **26** (1998) 117-133.
3. M. J. Berger and P. Colella, Local adaptive mesh refinement for shock hydrodynamics, J. Comput. Phys. **82** (1989) 64-84.
4. K. E. Brenan, S. L. Campbell and L. R. Petzold, Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, Second Edition, SIAM, 1995.
5. P. N. Brown, A. C. Hindmarsh and L. R. Petzold, *Using Krylov methods in the solution of large-scale differential- algebraic systems*, SIAM J. Sci. Comput., **15** (1994) 1467-1488.
6. Y. Cao, S. Li, L. Petzold, Adjoint sensitivity analysis for differential-algebraic equations: Part II, Numerical Solution, submitted.
7. Y. Cao, S. Li, L. Petzold and R. Serban, *Adjoint sensitivity analysis for differential-algebraic equations: Part I, The adjoint DAE system*, submitted.
8. J. E. Flaherty, P. K. Moore and C. Ozturan, Adaptive overlapping grid methods for parabolic systems, in Adaptive Methods for Partial Differential Equations, J. E. Flaherty, P.J. Paslow, M. S. Shephard, and J. D. Vasilakis, eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1989.
9. J. M. Hyman and S. Li, Solution Adapted Nested Grid Refinement for 2-D PDEs, Los Alamos National Lab. Report, LA-UR-98-5463 (1998).
10. J. M. Hyman, S. Li and L. R. Petzold, An Adaptive Moving Mesh Method with Static Rezoning for Partial Differential Equations, Los Alamos National Laboratory Report (1998).
11. S. Li and L. R. Petzold, Software and algorithms for sensitivity analysis of large-scale differential-algebraic systems, J. Comp. and Appl. Math., **125** (2001) 131-145.
12. S. Li and L. R. Petzold, Adjoint sensitivity analysis for partial differential-algebraic equations, in preparation.
13. J. Quirk, An Adaptive Grid Algorithm for Computational Shock Hydrodynamics, Ph.D thesis (1991), College of Aeronautics, Cranfield Institute of Tech.
14. Y. Saad, Iterative Methods for Sparse Linear Systems, PWS Publishing Company, Boston, 1996.
15. J. G. Verwer, J. G. Blom, VLUGR2: A Vectorized Local Uniform Grid Refinement Code for PDEs in 2D, Report NM-R9307 (1993), CWI, Amsterdam.
16. P. A. Zegeling, Moving Finite-Element Solution of Time-Dependent Partial Differential Equations in Two Space Dimensions. Department of Numerical Mathematics, CWI, Amsterdam, Report NM-R9206 (1992).